Program Analysis Techniques for Efficient Software Model Checking

Final Report
AFOSR Grant FA9550-07-1-0077
Program Manager: Robert L. Herklotz

PI: Chandrasekhar Boyapati Co-PI: Karem Sakallah

Electrical Engineering and Computer Science Department
University of Michigan, Ann Arbor, MI 48109
{bchandra, karem}@eecs.umich.edu

Abstract

The need to build next generation air force systems with highly complex functions, but at relatively low cost, will inevitably means a major investment in software. Without highly reliable software, any ambitious air force program cannot succeed. Indeed, software is the keystone (or perhaps the Achilles heel) of most large-scale automation projects; and the problem of making software reliable has become one of today's most important technological challenges.

To address this problem and to improve software reliability, we designed novel program analysis techniques that significantly speed up software model checking, thereby enabling the checking of much larger programs and broader class of program properties than previously possible.

In particular, we developed a software model checker for efficiently checking data oriented programs with respect to complex data dependent properties. We used our model checker for checking programs that use linked data structures such as lists, queues, trees, and maps. Verifying such programs has often been an obstacle to progress in the past and is a key underlying technical challenge in software verification. Because these programs have complex data dependent properties, the state space reduction techniques (such as predicate abstraction or partial order reduction) used by other model checkers are largely ineffective on such programs. Our model checker uses novel techniques to achieve orders of magnitude state space reduction.

In addition, we also developed a novel *trace driven* approach to use counter example guided abstraction refinement (CEGAR) to check for concurrency errors in multithreaded programs.

1 Introduction

Context

The motivation behind this research is the need for reliable and secure software. Software has become pervasive in civilian and military infrastructure. All activities including transportation,

telecommunications, energy, medicine, and banking rely on the correct working of software systems. Consequently, the problem of making software reliable and secure has become one of today's most important challenges. Multi-hundred-million-dollar space projects are interrupted by software glitches, power-grid failures are caused by bugs in software, and new security exploits are announced daily. Software reliability is crucial in critical systems, where failures can lead to loss of life—with risks ranging from a few individuals (anti-lock braking systems and airbag-deployment systems) to a few hundred (aircraft collision-avoidance systems) to tens of thousands (nuclear reactors and weapons systems). Software reliability also impacts security because buggy code underlies most security violations and progress in making systems more reliable will almost certainly make them more resistant to deliberate attack as well. Moreover, software reliability has a significant impact on economy. Studies estimate that bugs in software cost businesses worldwide about $175 billion [40] annually. Improving software reliability and security is thus essential and better tools and technologies are needed for identifying bugs and vulnerabilities in programs.

Air Force Context

The need to build the next generation autonomous and semi-autonomous air force systems with highly complex functions, but at relatively low cost, will inevitably mean a major investment in software. Already, software accounts for more than 60% of the cost of air force systems, and the cost of verification and validation of software sometimes comprises over 50% of the software development cost. These percentages will be even higher if the next generation systems are built using current software development and verification technologies because of the increase in the size and complexity of the software due to added functionality. But without highly reliable software, any ambitious defense program cannot succeed. Indeed, software is the keystone (or perhaps the Achilles heel) of most large-scale automation projects. One cannot over-emphasize the importance of this issue, especially in view of the reliability/delays/budget-overrun problems that have occurred in highly visible DoD projects, such as F/A-22 and SBIRS-HIGH.

Approach and Outline

Our research improves software reliability and security by enhancing the state of art in software model checking, thereby enabling the checking of much larger programs and broader class of program properties than previously possible. The rest of the report summarizes the main contributions of our research. More details about our research can be found in our publications [13, 37, 38, 39, 45, 46, 47] and a forthcoming Ph.D. thesis [36].

2 Glass Box Software Model Checking

Model checking is a formal verification technique that exhaustively tests a circuit/program on all possible inputs (usually up to a given size) and on all possible nondeterministic schedules. For hardware, model checkers have successfully verified fairly complex finite state control circuits with up to a few hundred bits of state information; but not circuits in general that have large data paths or memories. Similarly, for software, model checkers have primarily verified control-oriented programs with respect to temporal properties; but not much work has been done to verify data-oriented programs with respect to complex data-dependent properties.

Thus, while there is much research on software model checkers [2, 4, 7, 10, 11, 16, 18, 41, 21, 30] and on state space reduction techniques for software model checkers such as partial order reduction [17, 18] and tools based on predicate abstraction [19] such as Slam [2], Blast [21], or

Magic [7], none of these techniques seem to be effective in reducing the state space of data-oriented programs. For example, predicate abstraction relies on alias analysis that is too imprecise.

To address this problem, we introduced *glass box* software model checking. Our checker incorporates novel techniques to identify similarities in the state space of a model checker and safely prune large numbers of redundant states without explicitly checking them. Thus, while traditional software model checkers such as Java PathFinder (JPF) [41] and CMC [30] separately check every reachable state within a state space, our glass box checker checks a (usually very large) set of similar states in each step. This leads to orders of magnitude speedups over previous approaches.

Consider checking that a red-black tree [12] implementation maintains the red-black tree invariants. Previous model checking approaches such as JPF [41, 26], CMC [30], Korat [4], or Alloy [22, 25] systematically generate all red-black trees (up to a given size n) and check every red-black tree operation (such as insert or delete) on every red-black tree. Since the number of red-black trees with at most n nodes is exponential in n, these systems take time exponential in n for checking a red-black tree implementation. Our system works as follows. Our checker detects that any red-black tree operation such as insert or delete touches only one path in the tree from the root to a leaf (and perhaps some nearby nodes). Our checker then determines that it is sufficient to check every operation on every unique tree path (and some nearby nodes), rather than on every unique tree. Since the number of unique red-black tree paths is polynomial in n, our checker takes time polynomial in n. This leads to orders of magnitude speedups over previous approaches.

In general, our system works as follows. Consider checking a file system implementation, as another example. As our checker checks a file system operation o (such as reading, writing, creating, or deleting a file or a directory) on a file system state s, it uses its analyses to identify other file system states s'_1, s'_2, ..., s'_k on which the operation o behaves similarly. Our analyses guarantee that if o executes correctly on s, then o will execute correctly on every s'_i. Our checker therefore does not need to check o on any s'_i once it checks o on s. It thus safely prunes those state transitions from its search space, while still achieving complete test coverage within the bounded domain.

We call this the glass box approach to software model checking because our checker analyzes the behavior of an operation to prune large portions of the search space. We tested our system on a variety of programs and compared our system to other state of the art model checkers including Blast [21], JPF [41], and Korat [4]. We found that our system is significantly more efficient for checking data-oriented programs and data-dependent properties.

Note that like most model checking techniques [4, 16, 18, 41, 30], our system (in effect) exhaustively checks all states in a state space within some finite bounds. While this does not guarantee that the program is bug free because there could be bugs in larger unchecked states, in practice, almost all bugs are exposed by small program states. This conjecture, known as the *small scope hypothesis*, has been experimentally verified in several domains [23, 29, 34]. Thus, exhaustively checking all states within some finite bounds generates a high degree of confidence that the program is correct (with respect to the properties being checked).

Compared to our system, formal verification techniques that use theorem provers [3, 24, 32] are fully sound. However, these techniques require significant human effort (in the form of loop invariants or guidance to interactive theorem provers). For example, an unbalanced binary search tree implemented in Java can be checked in our system with less than 20 lines of extra Java code, implementing an abstraction function and a representation invariant. In fact, it is considered a good

programming practice [28] to write these functions anyway, in which case our system requires no extra human effort. However, checking a similar program using a theorem prover such as Coq [3] requires more than 1000 lines of extra human effort.

Compared to our system, other model checkers are more automatic because they do not require abstraction functions and representation invariants. However, our system is significantly more efficient than other model checkers for checking certain kinds of programs and program properties.

We present glass box software model checking as a middle ground between automatic model checkers and program verifiers based on theorem provers that require extensive human effort.

More details on this research can be found in [13].

3 Modular Glass Box Software Model Checking

To further improve the scalability of glass box software model checking, we introduced PIPAL, a system for *modular* glass box software model checking. In a modular checking approach program modules are replaced with *abstract implementations*, which are functionally equivalent but vastly simplified versions of the modules. The problem of checking a program then reduces to two tasks: checking that each program module behaves the same as its abstract implementation, and checking the program with its program modules replaced by their abstract implementations [9].

Extending traditional model checking to perform modular checking is trivial. For example, Java PathFinder (JPF) [41] or CMC [30] can check that a program module and an abstract implementation behave the same on every sequence of inputs (within some finite bounds) by simply checking every reachable state (within those bounds).

However, it is nontrivial to extend glass box model checking to perform modular checking, while maintaining the significant performance advantage of glass box model checking over traditional model checking. In particular, it is nontrivial to extend glass box checking to check that a module and an abstract implementation behave the same on every sequence of inputs (within some finite bounds). This is because, unlike traditional model checkers such as Java PathFinder or CMC, our glass box model checker does not check every reachable state separately. Instead it checks a (usually very large) set of similar states in each single step. Our research solves this problem.

We tested PIPAL on a variety of programs. Our experiments indicate that the modular model checking technique is far more efficient than checking programs as a unit.

More details on this research can be found in [37].

4 Glass Box Software Model Checking of Soundness of Type Systems

In addition to checking program properties, we also used our system on an orthogonal but interesting problem—of automatically checking soundness of type systems.

Type systems provide significant software engineering benefits. Types can enforce a wide variety of program invariants at compile time and catch programming errors early in the software development process. Types serve as documentation that lives with the code and is checked throughout the evolution of code. Types also require little programming overhead and type checking is fast and scalable. For these reasons, type systems are the most successful and widely used formal methods for detecting programming errors. Types are written, read, and checked routinely as part

of the software development process. However, the type systems in languages such as Java, C#, ML, or Haskell have limited descriptive power and only perform compliance checking of certain simple program properties. But it is clear that a lot more is possible. There is therefore plenty of research interest in developing new type systems for preventing various kinds of programming errors [6, 14, 20, 31, 42].

A formal proof of type soundness lends credibility that a type system does indeed prevent the errors it claims to prevent, and is a crucial part of type system design. At present, type soundness proofs are mostly done on paper, if at all. These proofs are usually long, tedious, and consequently error prone. There is therefore a growing interest in machine checkable proofs of soundness [1]. However, both the above approaches—proofs on paper (e.g., [15]) or machine checkable proofs (e.g., [33])—require significant manual effort.

Our research presents an alternate approach for checking type soundness *automatically* using a software model checker. Our idea is to systematically generate every type correct intermediate program state (within some finite bounds), execute the program one small step forward if possible using its small step operational semantics, and then check that the resulting intermediate program state is also type correct—but do so efficiently by detecting similarities in this search space and pruning away large portions of the search space. Thus, given only a specification of type correctness and the small step operational semantics for a language, our system automatically checks type soundness by checking that the progress and preservation theorems [35, 44] hold for the language (albeit for program states of at most some finite size).

Our experimental results on several languages—including the language of integer and boolean expressions from [35, Chapters 3 & 8], a typed version of the imperative language IMP from [43, Chapter 2], an object-oriented language which is a subset of Java, and a language with ownership types [5, 8]—indicate that our approach is feasible and that our search space pruning techniques do indeed significantly reduce what is otherwise an extremely large search space. Our research thus offers a promising approach for checking type soundness automatically, thereby enabling the design of novel type systems. In particular, this can enormously help programming language designers in debugging their language specifications. Currently there is no other technology around to automate this task effectively.

More details on this research can be found in [38].

5 Model Checking Multithreaded Programs Using Counter Example Guided Abstraction Refinement

Making multithreaded programming easier and less error-prone is an area of growing interest because of the increasing availability of inexpensive multicore hardware. In addition to the glass box software model checking, we also developed a novel approach to use counter example guided abstraction refinement or *CEGAR* [27] to check for concurrency errors in multithreaded programs. CEGAR creates and checks an abstraction of a program to reduce the state space. Abstractions that are too coarse generate counter examples. CEGAR uses them to refine the abstraction and redo the checking. We developed an efficient symbolic encoding of multithreaded programs and a novel trace driven abstraction and refinement approach to check their execution.

More details on this research can be found in [39, 45, 46, 47].

References

[1] B. E. Aydemir et al. Mechanized metatheory for the masses: The POPLMARK challenge, May 2005. http://www.cis.upenn.edu/ plclub/wiki-static/poplmark.pdf.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[3] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer Verlag, 2004.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[5] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, January 2003.

[6] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering (ICSE)*, June 2003.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[9] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: Algorithmic verification and debugging. *Communications of the ACM (CACM) 52(11)*, 2009.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[11] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *International Conference on Software Engineering (ICSE)*, June 2000.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1991.

[13] P. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2006.

[14] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[15] S. Drossopoulou and S. Eisenbach. Java is type safe—probably. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1997.

[16] M. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *Computer Aided Verification (CAV)*, January 2005.

[17] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages (POPL)*, January 2005.

[18] P. Godefroid. Model checking for programming languages using VeriSoft. In *Principles of Programming Languages (POPL)*, January 1997.

[19] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, June 1997.

[20] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[21] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Principles of Programming Languages (POPL)*, January 2002.

[22] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[23] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering (TSE) 22(7)*, July 1996.

[24] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[25] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. In *Automated Software Engineering (ASE)*, November 2001.

[26] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, April 2003.

[27] R. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoritic Approach*. Princeton University Press, 1999.

[28] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[29] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report TR-921, MIT Laboratory for Computer Science, September 2003.

[30] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Operating System Design and Implementation (OSDI)*, December 2002.

[31] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Principles of Programming Languages (POPL)*, January 1999.

[32] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002.

[33] T. Nipkow and D. von Oheimb. Java light is type-safe—definitely. In *Principles of Programming Languages (POPL)*, January 1998.

[34] J. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, October 2000.

[35] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[36] M. Roberson. Glass box software model checking. Ph.D. thesis, University of Michigan, Expected in May 2011.

[37] M. Roberson and C. Boyapati. Efficient modular glass box software model checking. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2010.

[38] M. Roberson, M. Harries, P. T. Darga, and C. Boyapati. Efficient software model checking of soundness of type systems. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2008.

[39] M. Said, L. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods Symposium (NFM)*, April 2011.

[40] The Sustainable Computing Consortium. http://www.sustainablecomputing.org.

[41] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Automated Software Engineering (ASE)*, September 2000.

[42] D. Walker. A type system for expressive security policies. In *Principles of Programming Languages (POPL)*, January 2000.

[43] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[44] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. In *Information and Computation 115(1)*, November 1994.

[45] Z. Yang, B. Al-Rawi, K. Sakallah, X. Huang, S. A. Smolka, and R. Grosu. Dynamic path reduction for software model checking. In *International Conference on Integrated Formal Methods (IFM)*, February 2009.

[46] Z. Yang and K. Sakallah. SMT-based symbolic model checking for multi-threaded programs. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly (EC^2)*, July 2008.

[47] Z. Yang and K. Sakallah. Trace-driven verication of multithreaded programs. In *International Conference on Formal Engineering Methods (ICFEM)*, November 2010.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
02-28-2011	Final Report	01-12-1006 to 30-11-2010

4. TITLE AND SUBTITLE
Program Analysis Techniques for Efficient Software Model Checking

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA9550-07-1-0077

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Chandrasekhar Boyapati
Karem Sakallah

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Michigan

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-OSR-VA-TR-2012-0076

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The need to build next generation air force systems with highly complex functions, but at relatively low cost, will inevitably means a major investment in software. Without highly reliable software, any ambitious air force program cannot succeed. Indeed, software is the keystone (or perhaps the Achilles heel) of most large-scale automation projects; and the problem of making software reliable has become one of today's most important technological challenges.

To address this problem and to improve software reliability, we designed novel program analysis techniques that significantly speed up software model checking, thereby enabling the checking of much larger programs and broader class of program properties than previously possible.

15. SUBJECT TERMS
Software Model Checking

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Chandrasekhar Boyapati
U	U	U	UU	8	19b. TELEPHONE NUMBER *(Include area code)* 734-763-9015

INSTRUCTIONS FOR COMPLETING SF 298

1. REPORT DATE. Full publication date, including day, month, if available. Must cite at least the year and be Year 2000 compliant, e.g. 30-06-1998; xx-06-1998; xx-xx-1998.

2. REPORT TYPE. State the type of report, such as final, technical, interim, memorandum, master's thesis, progress, quarterly, research, special, group study, etc.

3. DATES COVERED. Indicate the time during which the work was performed and the report was written, e.g., Jun 1997 - Jun 1998; 1-10 Jun 1996; May - Nov 1998; Nov 1998.

4. TITLE. Enter title and subtitle with volume number and part number, if applicable. On classified documents, enter the title classification in parentheses.

5a. CONTRACT NUMBER. Enter all contract numbers as they appear in the report, e.g. F33615-86-C-5169.

5b. GRANT NUMBER. Enter all grant numbers as they appear in the report, e.g. AFOSR-82-1234.

5c. PROGRAM ELEMENT NUMBER. Enter all program element numbers as they appear in the report, e.g. 61101A.

5d. PROJECT NUMBER. Enter all project numbers as they appear in the report, e.g. 1F665702D1257; ILIR.

5e. TASK NUMBER. Enter all task numbers as they appear in the report, e.g. 05; RF0330201; T4112.

5f. WORK UNIT NUMBER. Enter all work unit numbers as they appear in the report, e.g. 001; AFAPL30480105.

6. AUTHOR(S). Enter name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. The form of entry is the last name, first name, middle initial, and additional qualifiers separated by commas, e.g. Smith, Richard, J, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES). Self-explanatory.

8. PERFORMING ORGANIZATION REPORT NUMBER. Enter all unique alphanumeric report numbers assigned by the performing organization, e.g. BRL-1234; AFWL-TR-85-4017-Vol-21-PT-2.

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES). Enter the name and address of the organization(s) financially responsible for and monitoring the work.

10. SPONSOR/MONITOR'S ACRONYM(S). Enter, if available, e.g. BRL, ARDEC, NADC.

11. SPONSOR/MONITOR'S REPORT NUMBER(S). Enter report number as assigned by the sponsoring/ monitoring agency, if available, e.g. BRL-TR-829; -215.

12. DISTRIBUTION/AVAILABILITY STATEMENT. Use agency-mandated availability statements to indicate the public availability or distribution limitations of the report. If additional limitations/ restrictions or special markings are indicated, follow agency authorization procedures, e.g. RD/FRD, PROPIN, ITAR, etc. Include copyright information.

13. SUPPLEMENTARY NOTES. Enter information not included elsewhere such as: prepared in cooperation with; translation of; report supersedes; old edition number, etc.

14. ABSTRACT. A brief (approximately 200 words) factual summary of the most significant information.

15. SUBJECT TERMS. Key words or phrases identifying major concepts in the report.

16. SECURITY CLASSIFICATION. Enter security classification in accordance with security classification regulations, e.g. U, C, S, etc. If this form contains classified information, stamp classification level on the top and bottom of this page.

17. LIMITATION OF ABSTRACT. This block must be completed to assign a distribution limitation to the abstract. Enter UU (Unclassified Unlimited) or SAR (Same as Report). An entry in this block is necessary if the abstract is to be limited.